Notes on Orders of Growth

Formal notation for big-O:

f(n) is O(g(n)) (pronounced "eff of en is Oh (of) gee of en" or "... big-Oh ..."

iff there exists a constant k such that,

for all n > n0,

f(n) <= k * g(n)

Note that this means that f(n) grows *no faster than* g(n). If g(n) is big and f(n) is small, this is true but not especially helpful. (For example, most useful computer programs are O(2^n), but they're also O(something a heck of a lot smaller), or they wouldn't be particularly effective.) Computer scientists use O when they actually *mean* Theta.

f(n) is Theta(g(n)) (pronounced "eff of en is theta (of) gee of en" or "... big-Theta ..."

iff there exist constants k1 and k2 such that,

for all n > n0,

k1 * g(n) <= f(n) <= k2 * g(n)

Since this is about how functions behave in the limit, it is sometimes called **"asymptotic analysis"**.

The most important orders of growth to recognize have names:

- O(1) - **constant**
- O(log n) - **logarithmic**
- O(n) - **linear**
- O(n^2) etc. - **polynomial** [Note this is the P in the classes P and NP]
- O(2^n) etc. - **exponential**

Next time we'll also see an important class of n log n algorithms....

Arrays and lists:

Arrays implement two operations, both in constant time:

- get_value(index) -> value
- put_value(index, value)

It is possible to add a separate length/last-index constant-time operation using one extra piece of storage (and requiring some extra computational overhead to maintain it).

(Linked) **Lists** implement the following operations in constant time:

- get_value(node) -> value
- get_next(node) -> node
- empty?(node) -> boolean

Using these definitions, access to the list is simply access to the first node in the list, although in many languages there is often a small amount of overhead data to allow for uniform treatment of empty lists.

It is possible to add a separate length constant-time operation using one extra piece of storage (and requiring some extra computational overhead to maintain it).

It is also possible to add back-pointers (making this a "doubly linked list") and/or an end pointer. In this case, the node also supports get_prev(node) -> node.

Both Arrays and Linked Lists can be used to implement other linear structures. For example, a **Stack** supports the following four operations:

- push(value) [modifies the stack so that value is on top or returns a new stack meeting this condition, depending on whether it's a purely functional implementation]
- top() -> value [returns the value on top of the stack]
- pop() [modifies the stack to remove the top value or returns a new stack meeting this condition, depending on whether it's a purely functional implementation]
- empty?() -> boolean [true if there are no elements in the stack]

Stacks are sometimes called "last-in, first out" -- or LIFO -- structures

1. Sketch/pseudocode an implementation of these stack procedures using either an array or linked list.

2. What is the cost of each of the stack operations in your implementation? You may use asymptotic categories.

3. [optional] Sketch/pseudocode an implementation of tree-walking using a stack. Draw a simple tree with values stored both at internal nodes and at the leaves. (I recommend a tree with at least 5 nodes.) Consider walking this tree inside a while loop (i.e., without using a recursive procedure). You may assume that each node of the tree supports the following operations:

- children(tree) -> list of (sub)trees [can be manipulated with get_value/get_next]
- leaf?(tree) -> boolean (true if tree has no children)

Write out the list of nodes visited in order.

A ***Queue*** is a "first-in, first out" structure (or FIFO). It supports the following four operations:

- enQ(value) [modifies the queue so that value is at the end (or returns a new stack meeting this condition)]
- first() -> value [returns the first value in the queue]
- deQ() [modifies the queue to remove the first value (or returns a new stack meeting this condition)]
- empty?() -> boolean [true if there are no elements in the queue]

4. Sketch/pseudocode an implementation of these queue procedures using either an array or linked list.

5. What is the cost of each of the stack operations in your implementation? You may use asymptotic categ2ories.

6.[optional] Sketch/pseudocode an implementation of tree-walking using a queue. Ideally walk the same tree you considered in question 3.

Again, write out the list of nodes visited in order.

Tradeoffs

You've now seen:

- insertion in an unordered collection
- deletion from an unordered collection
- insertion and deletion from collections ordered by
    - recency of arrival (LIFO, FIFO) – these are above
    - some ordering operation <
- membership in unordered and ordered collections

7. Make a table of data structures vs. costs of these operations.

	insert	delete	member?
array, unordered			
array, ordered			
linked list, unordered			
linked list, ordered			

Consider best case, worst case, average; for member? also consider found and not found.

8. Pick a structure in the table above. Under what circumstances (what mixture of expected operations) would this be a good choice of data structure? Under what circumstances would it be a poor choice? What would be a better choice in these circumstances?

Searching revisited

We can do better if an ordered collection is stored in a random-access structure. This involves eliminating more than one stored value at a time.

8. How can we eliminate multiple values at once? What is the largest number of values we can eliminate with a single unit of cost? How many units will it take to find our value (or demonstrate that it is not there)?

9. Random access structures are inexpensive to search in this way, but they're expensive to insert into and delete from. We can get cheaper insertion/deletion if we use a pointer-based structure. What pointer-based structure (that we've already seen) echoes the search properties of the previous question?